This concept becomes a lot more interesting when complex patterns are employed, such as one that validates e-mail addresses: `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$`. Programmatically validating if an e-mail address is well-formed would require a great deal of code, while all the work can be done with a single regular expression in pattern matching.

PCRE syntax

The syntax that Nginx employs originates from the **Perl Compatible Regular Expression** (**PCRE**) library, which (if you remember *Chapter 2*, *Basic Nginx Configuration*) is a pre-requisite for making your own build, unless you disable the modules that make use of it. It's the most commonly used form of regular expressions, and nearly everything you learn here remains valid for other language variations.

In its simplest form, a pattern is composed of one character, for example, `x`. We can match strings against this pattern. Does `example` match the pattern `x`? Yes, `example` contains the character `x`. It can be more than one specific character—the pattern `[a-z]` matches any character between `a` and `z`, or even a combination of letters and digits: `[a-z0-9]`. In consequence, the pattern `hell[a-z0-9]` validates the following strings: `hello` and `hell4` but not `hell` or `hell!`.

You probably noticed that we employed the brackets `[` and `]`. They are part of what we call *metacharacters* and have a special effect on the pattern. There are a total of 11 metacharacters, and all play a different role. If you want to create a pattern that actually contains one of these characters, you need to escape the character with a `\` (backslash).

Metacharacter	Description
`^` Beginning	The entity after this character must be found at the beginning. Example pattern: `^h` Matching strings: `hello`, `h`, `hh` (anything beginning with h) Non-matching strings: `character`, `ssh`
`$` End	The entity before this character must be found at the end. Example pattern: `e$` Matching strings: `sample`, `e`, `file` (anything ending with e) Non-matching strings: `extra`, `shell`
`.` (dot) Any	Matches any character. Example pattern: `hell.` Matching strings: `hello`, `hellx`, `hell5`, `hell!` Non-matching strings: `hell`, `helo`

Metacharacter	Description
`[ ]` Set	Matches any character within the specified set. Syntax: `[a-z]` for a range, `[abcd]` for a set, and `[a-z0-9]` for two ranges. Note that if you want to include the – character in a range, you need to insert it right after `[` or just before `]`. Example pattern: `hell[a-y123-]` Matching strings: `hello`, `hell1`, `hell2`, `hell3`, `hell-` Non-matching strings: `hellz`, `hell4`, `heloo`, `he-llo`
`[^ ]` Negate set	Matches any character that is not within the specified set. Example pattern: `hell[^a-np-z0-9]` Matching strings: `hello`, `hell!` Non-matching strings: `hella`, `hell5`
`|` Alternation	Matches the entity placed either before or after `|`. Example pattern: `hello|welcome` Matching strings: `hello`, `welcome`, `helloes`, `awelcome` Non-matching strings: `hell`, `ellow`, `owelcom`
`( )` Grouping	Groups a set of entities, often used in conjunction with `|`. Also *captures* the matched entities; captures are detailed further on. Example pattern: `^(hello|hi) there$` Matching strings: `hello there`, `hi there`. Non-matching strings: `hey there`, `ahoy there`
`\` Escape	Allows you to escape special characters. Example pattern: `Hello\.` Matching strings: `Hello.`, `Hello. How are you?`, `Hi! Hello...` Non-matching strings: `Hello`, `Hello! how are you?`

Quantifiers

So far, you are able to express simple patterns with a limited number of characters. Quantifiers allow you to extend the number of accepted entities:

Quantifier	Description
`*` 0 or more times	The entity preceding `*` must be found 0 or more times. Example pattern: `he*llo` Matching strings: `hllo`, `hello`, `heeeello` Non-matching strings: `hallo`, `ello`

Quantifier	Description
`+` 1 or more times	The entity preceding `+` must be found 1 or more times. Example pattern: `he+llo` Matching strings: `hello`, `heeeello` Non-matching strings: `hllo`, `helo`
`?` 0 or 1 time	The entity preceding `?` must be found 0 or 1 time. Example pattern: `he?llo` Matching strings: `hello`, `hllo` Non-matching strings: `heello`, `heeeello`
`{x}` x times	The entity preceding `{x}` must be found x times. Example pattern: `he{3}llo` Matching strings: `heeello`, `oh heeello there!` Non-matching strings: `hello`, `heello`, `heeeello`
`{x,}` At least x times	The entity preceding `{x,}` must be found at least x times. Example pattern: `he{3,}llo` Matching strings: `heeello`, `heeeeeeello` Non-matching strings: `hllo`, `hello`, `heello`
`{x,y}` x to y times	The entity preceding `{x,y}` must be found between x and y times. Example pattern: `he{2,4}llo` Matching strings: `heello`, `heeello`, `heeeello` Non-matching strings: `hello`, `heeeeello`

As you probably noticed, the { and } characters in the regular expressions conflict with the block delimiter of the Nginx configuration file syntax language. If you want to write a regular expression pattern that includes curly brackets, you need to place the pattern between quotes (single or double quotes):

```
rewrite hel{2,}o /hello.php; # invalid
rewrite "hel{2,}o" /hello.php; # valid
rewrite 'hel{2,}o' /hello.php; # valid
```

Captures

One last feature of the regular expression mechanism is the ability to capture sub-expressions. Whatever text is placed between the parentheses `( )` is captured and can be used after the matching process. The captured characters become available under the form of variables called `$N`, where N is a numeric index, in order of capture. Alternatively, you can attribute an arbitrary name to each of your captures (see the next example). The variables generated through the captures can be inserted within the directive values. The following are a couple of examples that illustrate the principle:

Pattern	Example of a matching string	Captured
`^(hello|hi) (sir|mister)$`	`hello sir`	`$1 = hello` `$2 = sir`
`^(hello (sir))$`	`hello sir`	`$1 = hello sir` `$2 = sir`
`^(.*)$`	`nginx rocks`	`$1 = nginx rocks`

`^(.{1,3})([0-9]{1,4})([?!]{1,2})$`	`abc1234!?`	`$1 = abc`
		`$2 = 1234`
		`$3 = !?`

Named captures are also supported through the following syntax: `?<name>`. Example: `^/(?<folder>[^/]+)/(?<file>.*)$`	`/admin/doc`	`$folder = admin` `$file = doc`

When you use a regular expression in Nginx, for example, in the context of a `location` block, the buffers that you capture can be employed in later directives:

```
server {
    server_name website.com;
    location ~* ^/(downloads|files)/(.*)$ {
        add_header Capture1 $1;
        add_header Capture2 $2;
    }
}
```